Asynchronous Nested Parallelism for Dynamic Applications in Distributed Memory

Ioannis Papadopoulos, Nathan Thomas, Adam Fidel, Dielli Hoxha,
Nancy M. Amato, and Lawrence Rauchwerger

Parasol Lab, Dept. of Computer Science and Engineering, Texas A&M University

Abstract. Nested parallelism is of increasing interest for both expressivity and performance. Many problems are naturally expressed with this divide-and-conquer software design approach. In addition, programmers with target architecture knowledge employ nested parallelism for performance, imposing a hierarchy in the application to increase locality and resource utilization, often at the cost of implementation complexity.
While dynamic applications are a natural fit for the approach, support for nested parallelism in distributed systems is generally limited to well-structured applications engineered with distinct phases of intra-node computation and inter-node communication. This model makes expressing irregular applications difficult and also hurts performance by introducing unnecessary latency and synchronizations. In this paper we describe an approach to asynchronous nested parallelism which provides uniform treatment of nested computation across distributed memory. This approach allows efficient execution while supporting dynamic applications which cannot be mapped onto the machine in the rigid manner of regular applications. We use several graph algorithms as examples to demonstrate our library's expressivity, flexibility, and performance.

Keywords: nested parallelism, asynchronous, isolation, graph, dynamic applications

1 Introduction

Writing parallel applications is difficult, and many programming idioms taken for granted in sequential computing are often unavailable. One of these tools, program composition via *nested function invocation*, is not present in many parallel programming models, at least not in a general form that is abstracted from the target architecture. Indeed, while *nested parallelism* is a natural way to express many applications, employing it is often constrained by the deep memory hierarchies and multiple communication models of modern HPC platforms.

While the efficient mapping of the application's hierarchy of algorithms onto the machine's hierarchy is important for performance, we believe requiring developers to explicitly coordinate this effort is overly burdensome. Furthermore, direct management leads to ad-hoc solutions that significantly decrease *software reuse*, which is key to addressing the difficulties of parallel programming.

This work describes the support for nested parallelism in the runtime system of STAPL [8], a generic library of components for parallel program composition. The STAPL-RTS [29] serves the higher levels of the library, providing a uniform interface for computation and communication across distributed systems, while internally using shared memory optimizations where possible.

In this paper, we show how this uniform interface extends to the creation of nested parallel sections that execute STAPL algorithms. These nested SPMD (Single Program Multiple Data) sections provide an isolated environment from which algorithms, represented as task dependence graphs, execute and potentially spawn further nested computation. Each of these sections can be instantiated on an arbitrary subgroup of processing elements (i.e., locations) across distributed memory.

While the STAPL-RTS supports collective creation of nested parallel sections, in this work we focus on the *one-sided* interface. The one-sided interface allows a local activity (e.g., visiting a vertex in a distributed graph) on a given location to *spawn* a nested activity (e.g., following all edges in parallel to visit neighbors). As we will show, both the creation and execution of this nested activity are asynchronous: calls to the STAPL-RTS are non-blocking and allow local activities to proceed immediately. Hence, the one-sided, asynchronous mechanism is particularly suitable for dynamic applications.

Nested sections are also used to implement *composed data structures* with data distributed on arbitrary portions of the machine. Together, this support for nested algorithms and composed, distributed containers provides an increased level of support for irregular applications over previous work. In the experimental results, we demonstrate how the algorithms and data interact in a STAPL program. We use a distributed graph container with vertex adjacency lists being stored in various distributed configurations. Without any changes to the graph algorithm, we are able to test a variety of configurations and gain substantial performance improvements (2.25x at 4K cores) over the common baseline configuration (i.e., sequential storage of edge lists).

Our contributions include:

Uniform nested parallelism with controlled isolation. Support for arbitrary subgroups of processing elements (i.e., locations) across distributed memory. The sections are logically isolated, maintaining the hierarchical structure of algorithms defined by the user. For instance, message ordering and traffic quiescence is maintained separately for each nested section.

Asynchronous, one-sided creation of parallel sections. The ability to asynchronously create nested parallel sections provides latency hiding which is important for scalability. We combine one-sided and asynchronous parallel section creation, presenting a simple and scalable nested parallel paradigm.

Use of STAPL-RTS to implement dynamic, nested algorithms. We use the primitives to implement several fundamental graph algorithms, and demonstrate how various distribution strategies from previous work can be generalized under a common infrastructure using our approach to nested parallelism.

2 Related Work

When introduced, nested parallelism was used primarily for expressiveness, as in NESL [4]. The NESL compiler applies flattening, transforming all nested algorithms to a flat data parallel version, a technique with performance limitations.

OpenMP [27] has had nested parallelism capabilities since its inception. There is some work on nested parallelism for performance [15]. However, the `collapse` keyword in OpenMP 3.0 that flattens nested parallel sections attests to the difficulty of gaining performance from nested parallelism in OpenMP.

Other parallel programming systems employ nested parallelism for performance. Users express algorithms using nested sections for the sole purpose of exploiting locality. Restrictions are often imposed to achieve performance, limiting expressiveness. MPI [25] allows creating new MPI communicators by partitioning existing ones or by spawning additional processes. This functionality can be used to map nested parallel algorithms to the machine, however it mostly suits static applications, as each process must know through which MPI communicator it should communicate at any given point in the program.

Several systems enhance the MPI approach, while simplifying the programming model. Neststep [23] is a language that extends the BSP (bulk synchronous parallel) model and allows the partitioning of the processing elements of a superstep to smaller subsets or subgroups that can call any parallel algorithm. These subgroups need to finish prior to the parent group continuing with the next superstep. UPC [14] and Co-Array Fortran [24] have similar restrictions.

Another common approach is to use MPI for the first level parallelism (distributed memory) and OpenMP for the shared memory parallelism [10, 33], leading to ad-hoc solutions with manual data and computation placement.

Titanium [22] and UPC++ [38] introduce the Recursive SPMD (RSPMD) model and provide subgrouping capabilities, allowing programmers to call parallel algorithms from within nested parallel sections that are subsets of the parent section. Similarly to Neststep, they also require that the nested sections finish before resuming work in the parent section.

The Sequoia [16] parallel programming language provides a hierarchical view of the machine, enforcing locality through the nested parallelism and thread-safety with total task isolation: tasks cannot communicate with other tasks and can only access the memory address space passed to them. This strong isolation, in conjunction with execution restrictions to allow compile-time scheduling of task scheduling and task movement, limits its usefulness in dynamic applications.

Several systems support task-based parallelism, allowing the user to spawn tasks from other tasks. The programmer can thus express nested parallelism with the system responsible for placement. These include Intel Thread Building Blocks [32] and Cilk [5]. Since task placement is done in absence of knowledge about locality, one of the benefits of nested parallelism is lost.

X10 [12], Habanero-Java [11], HPX [20], and Fortress [21] all offer task-based parallelism, going a step further and allowing control over task placement. However, they suffer from loss of structure of the execution of the algorithms, as tasks are independent of each other. Building on top of Habanero, Otello [37]

addresses the issue of isolation in nested parallelism. While maintaining a task parallel system, Otello protects shared data structures through analysis of which object each task operates on and the spawning hierarchy of tasks.

Chapel [9] is a multi-paradigm parallel programming language and supports nested parallelism. While it supports data and task placement, users are given only two parallel algorithms (parallel for, reduce). Other parallel algorithms have to be implemented explicitly using task parallelism.

Legion [3] retains Sequoia's strong machine mapping capabilities while relaxing many of the assumptions of Sequoia, making it a good fit for dynamic applications. It follows a task parallel model in which tasks can spawn subtasks with controlled affinity. However, this process leads to loss of information about the structure of the parallel sections, as with other task parallel systems.

From Trilinos [2], Kokkos supports nested parallelism by allowing the division of threads in a team. Teams can be further divided and threads that belong to a team are concurrent. However, teams cannot execute concurrently, and only three algorithms (parallel for, reduce and scan) are available to be invoked from within a nested parallel section.

Phalanx [19] can asynchronously spawn SPMD tasks that execute on multiple threads. Programmers allocate memory explicitly on supported devices (CPU, GPU, etc.) and invoke tasks on them, creating parallel sections. Phalanx has a versatile programming model and is the most similar related work to the STAPL-RTS. Its main difference from the STAPL-RTS is that Phalanx requires explicit control of resources. Data and task placement needs to be statically specialized with the target (e.g., GPU, thread, process), transferring the responsibility of resource management to the user and creating the need for multi-versioned code.

3 STAPL Overview

The *Standard Template Adaptive Parallel Library* (STAPL) [8] is a framework developed in C++ for parallel programming. It follows the generic design of the Standard Template Library (STL) [26], with extensions and modifications for parallelism. STAPL is a library, requiring only a C++ compiler (e.g., `gcc`) and established communication libraries such as MPI. An overview of its major components are presented in Figure 1.

STAPL provides *parallel algorithms* and *distributed data structures* [34, 18] with interfaces similar to the STL. Instead of iterators, algorithms are written with *views* [7] that decouple the container interfaces from the underlying storage. The *skeletons framework* [36] allows the user to express an application as a composition of simpler parallel patterns (e.g., map, reduce, scan and others).

Algorithmic skeletons are instantiated at runtime as task dependence graphs by the `PARAGRAPH`, STAPL's data flow engine. It enforces task dependencies and is responsible for the transmission of intermediate values between tasks.

The runtime system (STAPL-RTS) [35, 29] provides portable performance by abstracting the underlying platform with the concept of *locations*. A *location* is a component of a parallel machine that has a contiguous memory address space

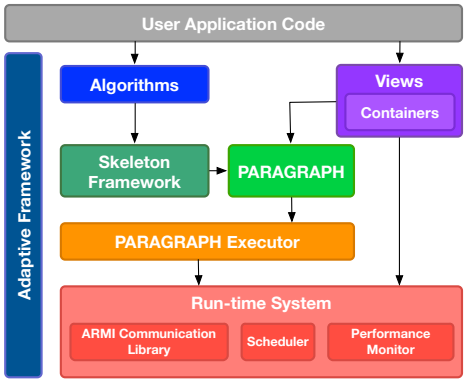


Fig. 1. STAPL Components

and has associated execution capabilities (e.g., threads). Locations only have access to their own address space and communicate with other locations using *Remote Method Invocations* (RMIs) on shared objects.

Containers and `PARAGRAPH`s are both distributed objects (i.e., `p_object`s). RMIs are used in the containers to read and write elements. RMIs are used in the `PARAGRAPH` to place tasks, resolve dependencies, and flow values between tasks that are not on the same location.

Each distributed object has an associated set of locations on which it is distributed. The STAPL-RTS abstracts the platform and its resources, providing a uniform interface for all communication in the library and applications built with it. This abstraction of a virtual distributed, parallel machine helps STAPL support general nested parallelism.

4 Asynchronous Nested Parallelism in STAPL

As with STL programs, a typical STAPL application begins with the instantiation of the necessary data structures. Each container has its own distribution and thus defines the affinity of its elements. Container composition is supported, as well complete control over the distribution of each container (e.g., balanced, block cyclic, arbitrary). For example, the case study presented in Section 5 uses composed instances of the STAPL `array` for vertex and edge storage, with various distribution strategies considered for each nested edge list. Users write applications with the help of skeletons [36] and views, that abstract the computation and data access, respectively. The views provide element locality information, projecting it from the underlying container.

An algorithm's execution is performed by a `PARAGRAPH`, a distributed task dependence graph responsible for managing task dependencies and declaring which tasks are runnable. Each `PARAGRAPH` executes in an isolated environment, with data access provided by the views. Each task may itself be a parallel algorithm, for which a new nested parallel section is created. A default policy

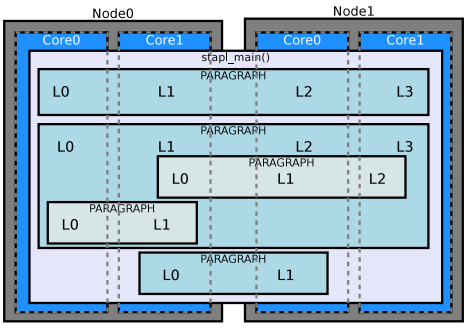


Fig. 2. Execution Model

places a `PARAGRAPH` for execution based on the locality of the data it accesses, and custom policies can passed to the `PARAGRAPH` at creation. Figure 2 shows an example execution instance of an application that has a number of `PARAGRAPH` invocations in isolated parallel sections over the same set of hardware resources.

4.1 STAPL Design Considerations

In order to take advantage of nested parallelism and realize its full potential, we have made several design decisions that influence our implementation including:

Expressiveness. STAPL users express algorithms as a composition of simpler parallel algorithms using algorithmic skeletons [36]. This specification is independent of any target architecture. The responsibility for mapping it onto the machine is left to the library, though it can be customized by more experienced users at an appropriate level of abstraction.

Preserving Algorithm Structure. We maintain the hierarchy of tasks defined by the application when mapping it to the machine. Hence, each nested section's tasks remain associated with it and are subject to its scheduling policy. Each algorithm invocation is run within an SPMD section, from which both point-to-point and collective operations are accounted for independent of other sections. The SPMD programming model has been chosen since scaling on distributed machines has favored this programming model (e.g., MPI [25]) more than fork-join or task parallel models.

Parallel Section Isolation. STAPL parallel sections exhibit controlled isolation for safety and correctness. The uncontrolled exchange of data between parallel sections is potentially unsafe due to data races. Performance can be impacted, as isolation means that collective operations and data exchanges are in a controlled environment. We discuss techniques to mitigate these overheads in [29]. Users provide views to define the data available for access in each section.

Asynchronous, One-sided Parallel Section Creation. We support both *partitioning* (collective creation) of existing environments and *spawning* (one-sided creation) of new environments. Partitioning existing parallel sections is

Name	SPMD NP sections	Asynchronous	Locality Aware	Any algorithm allowed in NP section
MPI	Yes	No	Manual	Yes
UPC++, Co-Array Fortran, Titanium	Yes	No	Manual	Yes
Sequoia	Yes	No	Compile-time	Yes
Habanero, X10	No	Yes	Yes	Yes
Chapel	No	Yes	Yes	No
Charm++	No	Yes	Yes	Yes
Legion	No	Yes	Yes	Yes
Phalanx	Yes	Yes	Manual	Yes
STAPL	Yes	Yes	Yes	Yes

Table 1. Nested Parallelism (NP) capabilities comparison

beneficial for static applications but is difficult to use in dynamic applications. On the other hand, one-sided creation may not give optimal performance for static applications where the structure of parallelism is more readily known.

In this paper, we only present the one-sided world creation, as the collective implementation is similar to other systems for subgrouping (e.g., Titanium, MPI and others). One-sided creation is fully asynchronous. This allows us to effectively hide latency and supports our always distributed memory model. Table 1 summarizes the main differences between our model and similar approaches.

4.2 Execution Model

The STAPL-RTS presents a *unified interface* for both intra-node and inter-node communication to support performance portability. Internally the *mixed-mode* implementation uses both standard shared and distributed memory communication protocols when appropriate. For scalability and correctness, we employ a *distributed Remote Method Invocation* (RMI) model.

Each processing element together with a logical address space forms an isolated computational unit called a *location*. Each location has an isolated, virtual address space which is not directly accessible by other locations. When a location wishes to modify or read a remote location's memory, this action must be expressed via RMIs on distributed objects, called `p_objects`.

Gangs represent STAPL-RTS subgroup support. Each gang is a set of N locations with identifiers in the range $[0, \ldots, N-1]$ in which an SPMD task executes. It has the necessary information for mapping its locations to processing elements and describing a topology for performing collective operations. While the locations of a gang execute a single SPMD task, they communicate asynchronously independently of each other, making them a more loosely knit group than for example MPI groups of Titanium/UPC teams. To create a new gang, one either:

- *Partitions* an existing gang with *collective gang creation* over the locations that participate in the new gang.

– *Spawns* a gang, whereby one location creates a new gang in an asynchronous and one-sided manner, using a subset of locations in an existing gang.

`p_object`s can be created within a gang, and as such, each `p_object` is associated with exactly one gang and is distributed across its locations. A gang can have any number of `p_object`s. Each `p_object` can be referenced either with a regular C++ reference inside the gang it was created or through handles.

4.3 One sided Gang creation

The STAPL-RTS provides primitives for the one-sided creation of gangs via allocating `p_object`s on a set of pre-existing locations. An example is shown in Figure 3. The first `construct()` call creates a new parallel section over the locations $\{0, 2, 4, 5, 6\}$ of the current section and creates an instance of `T`. The second `construct()` call creates an object of type `U` in a new gang that is co-located with the gang of the previous object.

Multiple variations are supported, such as creating gangs on arbitrary ranges of locations (or all) of either the current parallel section or that of another `p_object`. The STAPL-RTS is responsible for translating location IDs to processing element (PE) IDs and for building a suitable multicast tree on the PEs which it uses to construct the gang and the associated `p_object`. We plan on extending this support to define gangs over specific parts of an hierarchical or heterogeneous machine, such as over a specific socket or accelerator.

```
1  using namespace stapl;
2
3  // Create a p_object of type T by passing args to the constructor, in a
       new gang over the given locations and return a future to its handle
4  future<rmi_handle::reference> f1 =
5    construct<T>(location_range, {0, 2, 4, 5, 6}, args...);
6
7  // Get object handle
8  auto h = f1.get();
9
10 // Create a new p_object of type U on a new gang co-located with the
       gang of the first object
11 future<rmi_handle::reference> f2 =
12   construct<U>(h, all_locations, args...);
```

Fig. 3. `construct()` example usage.

A gang's lifetime is tied to that of the `p_object`s present in it. Figure 4 is the state transition diagram of the life of a gang.

– Upon construction, the gang is *created*. The necessary metadata is generated and everything is set up to execute the SPMD task.
– When the task executes, the gang is declared *running*. While the task executes, `p_object`s can be created and they are automatically associated with

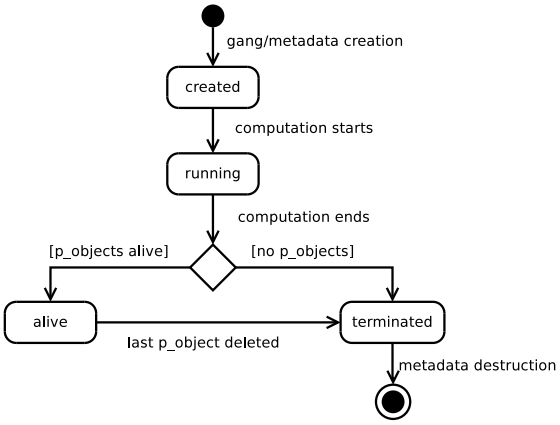


Fig. 4. Gang State Transition Diagram

the gang. The scope of the automatic `p_objects` (stack allocated) is the scope of the SPMD task, however heap-allocated `p_objects` can outlive it.
– If the task finishes and there are no associated `p_objects`, the gang is *terminated* and its metadata is deleted.
– If there are still `p_objects` associated with the gang, then it is declared *alive* and its metadata preserved. The gang remains alive until the last `p_object` is deleted. RMIs can still be invoked on the `p_objects`.

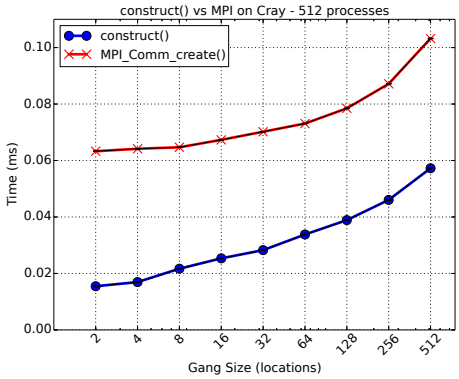


Fig. 5. `construct()` vs MPI on 512 processes on a Cray XK7m-200

Figure 5 presents a micro benchmark of the `construct` primitive on a Cray XK7m-200 (described in Section 5.1). It compares our range-based `construct()` against `MPI_Comm_create()` over the same number of processes, when the global parallel section is 512 processes. The combined effect of asynchronous creation

and deletion, as well as the fact that the MPI primitive is collective over the set of locations, while our primitive is one-sided, result in competitive performance against MPI and shows that it is a scalable approach.

5 Case Study - Graph Algorithms

Processing large-scale graphs has become a critical component in a variety of fields, from scientific computing to social analytics. An important class of graphs are *scale-free* networks, where the vertex degree distribution follows a power-law. These graphs are known for the presence of *hub vertices* that have extremely high degrees and present challenges for parallel computations.

In the presence of hub vertices, simple 1D partitioning (i.e., vertices distributed, edges colocated with corresponding vertex) of scale-free networks presents challenges to balancing per processor resource utilization, as the placement of a hub could overload a processor. More sophisticated types of partitioning have been proposed, including checkerboard 2D adjacency matrix partitioning [6], edge list partitioning [30] and specialized techniques for distributing hub vertices [31, 17]. However, these strategies often change both the data representation as well as the algorithm, making it difficult to unify them in a common framework.

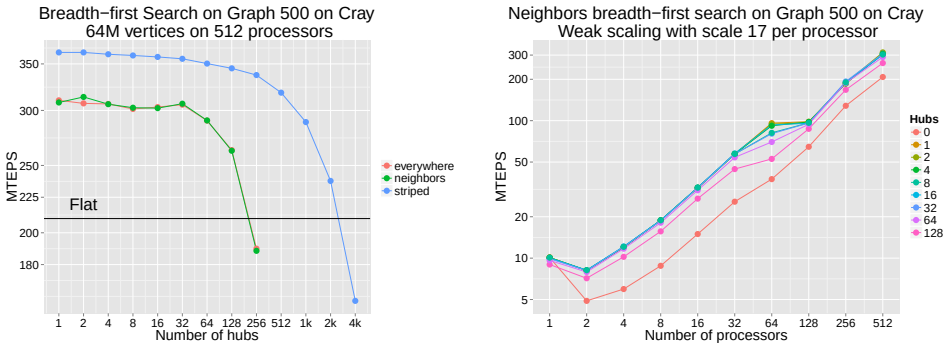
We represent the graph as a distributed array of vertices, with each vertex having a (possibly) distributed array of edges. Using `construct` we define several strategies for distributing the edges of hub vertices, that can be interchanged without changing the graph algorithm itself. The first distribution strategy (`EVERYWHERE`) places a hub's adjacency list on all locations of the graph's gang. The second (`NEIGHBORS`) places the edges only on locations where the hub has neighbors. This strategy is especially dynamic as the distribution of each hub edge list is dependent on the input data. Thus, we rely heavily on the arbitrary subgroup support of STAPL-RTS. The last strategy (`STRIPED`) distributes the adjacency list on one location per shared-memory node in a strided fashion to ensure that no two hubs have edges on the same location.

Even though the distribution strategy of the edges changes, the edge visit algorithm remains unchanged; the `PARAGRAPH` executing the algorithm queries the edge view about the locality of the underlying container and transparently spawns the nested section onto the processing elements where locations in the container's gang are present. This one-sided, locality driven computational mapping is a natural fit for the application and allows easily experimentation with novel and arbitrary mappings of the edges to locations, without the overhead of rewriting and hand-tuning the algorithm to support these changes.

5.1 Experimental Setup

We performed our experiments on two different systems. The code was compiled with maximum optimization levels (`-DNDEBUG -O3`).

`Cray` is a Cray XK7m-200 with twenty-four compute nodes of 2.1 GHz AMD Opteron Interlagos 16-core processors. Twelve nodes are single socket with 32 GB RAM, and twelve are dual socket with 64 GB RAM. The compiler was `gcc` 4.9.2.



(a) (b)

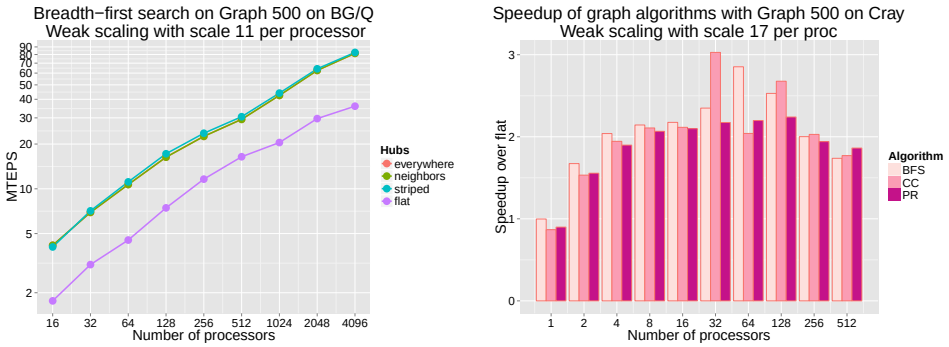
Fig. 6. Graph 500 breadth-first search on Cray varying (a) the number of hubs on 512 processors and (b) the number of processors for a weak scaling experiment.

`BG/Q` is an IBM BG/Q system at Lawrence Livermore National Laboratory. It has 24,576 nodes, each with a 16-core IBM PowerPC A2 processor at 1.6 GHz and 16 GB of RAM. The compiler used was `gcc` 4.7.2.

5.2 Experimental Evaluation

To validate our approach, we implemented the Graph 500 benchmark [1], which performs a parallel breadth-first search on a scale-free network. Figure 6(a) shows the breadth-first search algorithm over the Graph 500 input graph. As shown, all three edge distribution strategies fare well over the baseline of non-distributed adjacency lists for modest number of hubs, and then degrade in performance as more vertices are distributed. The `EVERYWHERE` and `NEIGHBORS` strategies behave similarly, as the set of locations that contain any neighbor is likely to be all locations for high-degree hub vertices. The `EVERYWHERE` and `NEIGHBORS` strategies are 49% and 51% faster than the baseline, respectively. The `STRIPED` strategy performs up to 75% faster than the baseline, which is a further improvement over the other strategies. On `Cray`, cores exhibit high performance relative to the interconnect, and thus even modest amounts of communication can bring about large performance degradation. The `STRIPED` strategy reduces the amount of off-node communication to create the parallel section from the source vertex location, bringing the performance of the algorithm above the other two strategies. We are investigating this phenomenon to derive a more rigorous model for distributing edge lists.

Figure 6(b) shows a weak scaling study of the neighbor distribution strategy on `Cray`. As shown, the flat breadth-first search scales poorly from 1 to 2 processors due to an increase in the amount of communication. By distributing the edges for hubs, we reduce this communication and provide better performance than the flat algorithm. The number of distributed hubs must be carefully chosen:



(a) (b)

Fig. 7. Graph 500 (a) breadth-first search with various adjacency distributions on BG/Q and (b) various graph analytics algorithms on Cray.

too few hubs will not provide sufficient benefit in disseminating edge traversals, whereas too many hubs could overload the communication subsystem.

In order to evaluate our technique at a larger scale, we evaluated breadth-first search on the Graph 500 graph on BG/Q in Figure 7(a). We found that although faster than the flat version, all three distribution strategies performed comparably with each other. At 4,096 processors, the distributed adjacency list versions of breadth-first search are 2.25x faster than the flat baseline. Hence, the distribution strategy is machine-dependent, further reinforcing the need for a modular and algorithm-agnostic mechanism to explore the possible configuration space for nested parallelism in parallel graph algorithms.

Finally, to show the generality of the nested algorithm support in the context of dynamic computations, we implement two other popular graph analytics algorithms: Hash-Min connected components (CC) [13] and PageRank [28] (PR). In Figure 7(b) we present the *oracle speedup* of the nested parallel versions over the flat version, where speedup is measured by computing the ratio between the best configuration and hub count for the nested parallel version and the flat version. All three algorithms show substantial improvement for all processor counts except for 1, where the overhead of creating a nested parallel section is measured. In some cases, the nested parallel version is able to achieve upwards of 3x speedup, such as on connected components at 32 cores.

6 Conclusion

In this paper we presented support for one-sided, asynchronous nested parallelism in STAPL-RTS. It is utilized in STAPL for the implementation of composed containers and the `PARAGRAPH` which manages algorithm execution. These components provide flexible support for nested parallelism, with intelligent placement of parallel sections based on the abstract locality information provided by

our runtime. We demonstrated the benefit of the approach via a case study of graph algorithms, where significant gains were attained by tuning the locality of the data structure *independent of the algorithm specification*.

For future work, we want to to implement other dynamic programs using the one-sided nested parallel constructs. We also plan to use our graph framework to explore other possible computation and data distribution strategies with the aim of performance portability. We think these nested parallelism constructs are applicable to a broad range applications, allowing STAPL to provide a high level of expressiveness, while still mapping efficiently onto large, distributed systems.

7 Acknowledgments

This research is supported in part by NSF awards CNS-0551685, CCF-0702765, CCF-0833199, CCF-1439145, CCF-1423111, CCF-0830753, IIS-0916053, IIS-0917266, EFRI–1240483, RI-1217991, by NIH NCI R25 CA090301-11, by DOE awards DE-AC02-06CH11357, DE-NA0002376, B575363, by Samsung, IBM, Intel, and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST). This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

1. The graph 500 list. `http://www.graph500.org`, 2011.
2. C. G. Baker and M. A. Heroux. Tpetra, and the use of generic programming in scientific computing. *Sci. Program.*, 20(2):115–128, Apr. 2012.
3. M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, Nov 2012.
4. G. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-93-129, Carnegie Mellon University, 1993.
5. R. D. Blumofe et al. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, volume 30, pages 207–216, New York, NY, USA, July 1995. ACM.
6. A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 65:1–65:12, New York, NY, USA, 2011. ACM.
7. A. Buss, A. Fidel, Harshvardhan, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL pView. In *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC), in Lecture Notes in Computer Science (LNCS)*, Houston, TX, USA, September 2010.
8. A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. STAPL: Standard template adaptive parallel library. In *Proc. Annual Haifa Experimental Systems Conference (SYSTOR)*, pages 1–10, New York, NY, USA, 2010. ACM.

9. D. Callahan, B. L. Chamberlain, and H. P. Zima. The Cascade High Productivity Language. In *The Ninth Int. Workshop on High-Level Parallel Programming Models and Supportive Environments*, volume 26, pages 52–60, Los Alamitos, CA, USA, 2004.
10. F. Cappello and D. Etiemble. Mpi versus mpi+openmp on ibm sp for the nas benchmarks. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.
11. V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: The New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.
12. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an Object-Oriented Approach to Non-Uniform Cluster Computing. In *Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.
13. L. Chitnis et al. Finding connected components in map-reduce in logarithmic rounds. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 50–61, Washington, DC, USA, 2013. IEEE Computer Society.
14. U. Consortium. *UPC Language Specifications V1.2*, 2005. `http://www.gwu.edu/~upc/publications/LBNL-59208.pdf`.
15. R. Duran et al. Runtime adjustment of parallel nested loops. In *In Proc. of the International Workshop on OpenMP Applications and Tools (WOMPAT 04*, 2004.
16. K. Fatahalian et al. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.
17. J. E. Gonzalez et al. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.
18. Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. The stapl parallel graph library. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 46–60. Springer Berlin Heidelberg, 2012.
19. T. D. R. Hartley et al. Improving Performance of Adaptive Component-based Dataflow Middleware. *Parallel Comput.*, 38(6-7):289–309, June 2012.
20. T. Heller et al. Using hpx and libgeodecomp for scaling hpc applications on heterogeneous supercomputers. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '13, pages 1:1–1:8, New York, NY, USA, 2013. ACM.
21. G. L. S. Jr. et al. Fortress (Sun HPCS Language). In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 718–735. Springer, 2011.
22. A. Kamil and K. A. Yelick. Hierarchical Computation in the SPMD Programming Model. In C. Cascaval and P. Montesinos, editors, *LCPC*, volume 8664 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2013.
23. C. W. Keßler. NestStep: Nested Parallelism and Virtual Shared Memory for the BSP Model. *The Journal of Supercomputing*, 17(3):245–262, 2000.
24. J. Mellor-Crummey et al. A new vision for coarray Fortran. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 5:1–5:9, New York, NY, USA, 2009. ACM.

25. MPI forum. MPI: A Message-Passing Interface Standard Version 3.1. `http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`, 2015.
26. D. Musser, G. Derge, and A. Saini. *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley, 2001.
27. OpenMP Architecture Review Board. OpenMP Application Program Interface. Specification, 2011.
28. L. Page et al. The pagerank citation ranking: Bringing order to the web. 1998.
29. I. Papadopoulos, N. Thomas, A. Fidel, N. M. Amato, and L. Rauchwerger. STAPL-RTS: an application driven runtime system. In *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015*, pages 425–434, 2015.
30. R. Pearce, M. Gokhale, and N. M. Amato. Scaling techniques for massive scale-free graphs in distributed (external) memory. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 825–836, Washington, DC, USA, 2013. IEEE Computer Society.
31. R. Pearce, M. Gokhale, and N. M. Amato. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 549–559, Piscataway, NJ, USA, 2014. IEEE Press.
32. J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.
33. J. Sillero et al. Hybrid openmp-mpi turbulent boundary layer code over 32k cores. In *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'11, pages 218–227, Berlin, Heidelberg, 2011. Springer-Verlag.
34. G. Tanase, A. Buss, A. Fidel, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Container Framework. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, pages 235–246, San Antonio, Texas, USA, 2011.
35. N. Thomas, S. Saunders, T. G. Smith, G. Tanase, and L. Rauchwerger. Armi: a high level communication library for stapl. *Parallel Processing Letters*, 16(2):261–280, 2006.
36. M. Zandifar, M. Abdul Jabbar, A. Majidi, D. Keyes, N. M. Amato, and L. Rauchwerger. Composing algorithmic skeletons to express high-performance scientific applications. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 415–424, New York, NY, USA, 2015. ACM. Conference Best Paper Award.
37. J. Zhao, R. Lublinerman, Z. Budimlić, S. Chaudhuri, and V. Sarkar. Isolation for Nested Task Parallelism. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 571–588, New York, NY, USA, 2013. ACM.
38. Y. Zheng et al. Upc++: A pgas extension for c++. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1105–1114, May 2014.